

Lab5 后端代码的生成

甘文迪 PB19030801

目标

将 `Light IR` 翻译成 `x86_64` 汇编指令，用 `clang` 产生可执行文件。

需要开启 `-mem2reg` 优化选项

设计

类

- `Position` 表示存储位置，主要种类有 `Register` , `MemoryAddress` , `ConstInteger`
- `AsmInstruction` 表示汇编语句
- `AsmBlock` 表示汇编语句块
- `AsmFunction` 表示汇编函数



```
int main(void) {  
    int a;  
    a = 1 + 2;  
    return a;  
}
```



AsmFunction

```
.text  
.globl main  
main:  
    >> .cfi_startproc  
    >> pushq >> %rbp  
    >> movq >> %rsp, %rbp  
    >> subq >> $16, %rsp  
    >> .main_label_entry:  
    >> movl >> -4(%rbp), %r10d  
    >> movl >> $1, %r10d  
    >> addl >> $2, %r10d  
    >> movl >> %r10d, -4(%rbp)  
    >> movl >> -4(%rbp), %eax  
    >> addq >> $16, %rsp  
    >> popq >> %rbp  
    >> retq  
    >> .Lfunc_end1:  
    >> .cfi_endproc
```

AsmBlock

基本的函数

```
void updateRegister(Value* value);           // 使用 LRU 更新寄存器状态
MemoryAddress& getAddress(Value* value);     // 获取地址，否则开辟地址
Register& getEmptyRegister(Value* value);    // 获得空的寄存器
Position& getPosition(Value* value);         // 寻找位置，优先给出常量、寄存器，否则取内存地址
```

寄存器分配

分别选择以下寄存器作为整数和浮点数变量的临时寄存器

```
r10d, r11d, r12d, r13d, r14d, r15d
xmm8, xmm9, xmm10, xmm11, xmm12, xmm13, xmm14, xmm15
```

当请求或使用寄存器时，用 `updateRegister` 进行更新。

当尝试用一个 `value` 去请求空的寄存器时，若寄存器存有其他值，则将其转移到内存后再请求。

各种指令的翻译

1. 运算

- 数值
- 比较
- 类型转换

2. 跳转及 phi

3. 函数

- 返回
- 函数起始部分
- 调用

4. 指针相关

1. 运算

数值运算

以整数加法为例

参考 clang -S 生成的代码

```
a = 1 + 2;
```

```
movl    $1, %r10d
addl    $2, %r10d
```

```
auto reg = getEmptyRegister(instruction);
appendInst(movl, getPosition(value1), reg);
appendInst(addl, getPosition(value2), reg);
```

汇编指令	含义
addl	整数加
subl	整数减
imull	整数乘
addss	浮点数加
subss	浮点数减
mulss	浮点数乘
divss	浮点数除

1. 浮点数常量的表示

函数开头部分加上

```
.main_0:  
    .long    1067030938
```

调用时使用 `.main_0(%rip)`

2. 整数除法

```
a = 9 / 4;
```



```
movq    $9, %rax  
movl    $4, %r10d  
cld  
idivl    %r10d  
movl    %eax, %r10d
```


比较运算

整数比较

2 > 1 翻译成

```
movl    $2, %eax
cmpl    $1, %eax
setg    %cl
movzbl  %cl, %r10d
```

浮点数比较

```
movss   .main_0(%rip), %xmm8
ucomiss .main_1(%rip), %xmm8
seta    %cl
movzbl  %cl, %r10d
```

零扩展 `zext` 部分不生成汇编代码

类型转换

fptosi

```
cvttss2si    %xmm8, %r10d
```

sitofp

```
movl    $1, %eax
cvtsi2ssl    %eax, %xmm8
```

2. 跳转及 phi

跳转语句

在每个 `AsmBlock` 前需加上编号

```
.main_label2:
```

无条件

```
br label %label1
```



```
jmp .main_label1
```

有条件

```
%op5 = icmp ne i32 %op4, 0  
br i1 %op5, label %label6, label %label9
```



```
cmpl    $0, %r12d  
jne     .main_label6  
jmp     .main_label9
```

phi 语句

直接在来自的基本块的末尾加指令存在问题。因为如果这个基本块在后方，还未进行处理，其中还没有指令，会在最前面生成指令。

因此增加 `endInstructions`，用于存储基本块的返回、跳转、phi 语句的翻译结果。翻译时在来自的基本块 `endInstructions` 的最前面插入 `movl` 指令。

```
.main_label_entry:
```

```
    movl    %eax, -4(%rbp)
    jmp     .main_label1
```

```
.main_label1:
```

```
    movl    -12(%rbp), %r10d
    movl    -4(%rbp), %eax
```

```
.main_label6:
```

```
    movl    %eax, -4(%rbp)
    jmp     .main_label1
```



```
label1:
```

```
    %op11 = phi i32 [ 10, %label_entry ], [ %op8, %label6 ]
```

3. 函数

- 返回语句 `ret`
- 函数起始部分
- 调用语句 `call` (详细)

返回语句

```
ret i32 3
```



```
movl    $3, %eax  
addq    $16, %rsp  
popq    %rbp  
retq
```

函数起始部分

处理栈指针

```
pushq   %rbp  
movq    %rsp, %rbp  
subq    $16, %rsp
```

将参数移入内存

临时空间的大小需要全部扫描后才能确定，最后再生成返回语句

调用语句

对于整数参数，参数优先进入寄存器 `edi, esi, edx, ecx, r8d, r9d`，否则压入栈

```
FOR (i, 1, operandNumber - 1) {
    Position& position = getPosition(operands[i]);
    auto type = operands[i]->get_type();
    if (type == int32Type) {
        if (intRegisterIndex < argIntRegister.size())
            appendInst(movl, position, *argIntRegister[intRegisterIndex++]);
        else
            appendInst(pushq, position);
    }
}
```

保存寄存器的值，然后调用函数

```
stash();  
appendInst(call, Position(callFunctionName));
```

获取返回值，扣除参数占用的栈空间

```
if (returnType == int32Type)  
    appendInst(movl, eax, getEmptyRegister(instruction));  
else if (returnType == floatType)  
    appendInst(movss, xmm0, getEmptyRegister(instruction));  
if (operandNumber >= 7)  
    appendInst(addq, ConstInteger(8 * (operandNumber - 7)), rsp); // pop
```

浮点数及指针类似

4. 指针相关

由于开启了 `-mem2reg` 优化选项，不考虑局部的指针变量

不同类型的数组或指针可用的指令

	数组	指针	全局变量
<code>alloca</code>	✓		
<code>load</code>		✓	✓
<code>store</code>		✓	✓
<code>getelementptr</code>	✓	✓	全局数组可以

数组

alloca

先获取空间，再设置值 `instruction` 对应指针的地址

```
if (allocaType->get_type_id() == Type::ArrayTyID) {  
    auto arrayType = static_cast<ArrayType*>(allocaType);  
    int nums = arrayType->get_num_of_elements();  
    stackSpace += nums * 4;  
}  
getAddress(instruction);
```

栈增长方向 ↑
指向数组的指针 (8 字节)
数组 (400 字节)

getelementptr

由于 cminus 的数组只有一维，且 `int` `float` 均占用 4 字节，数组和指针均可用

`ans = pointer + 4 * index` 来计算偏移。

```
movq    -416(%rbp), %r11
movl    $1, %eax
imull    $4, %eax
addq    %rax, %r11
```

`%op8 = getelementptr [100 x i32], [100 x i32]* %op0, i32 0, i32 1`

指针

load

用 `%rax` 临时存储地址，用 `0(%rax)` 获取值

```
%op9 = load i32, i32* %op8
```



```
movq    -16(%rbp), %rax
movq    0(%rax), %r11
```

`store` 语句同理

全局变量

需要在生成的代码开始处添加 `.comm`

```
.comm    x, 400, 4
```

为了统一非数组和数组，在**函数开始处**获取其存储的地址，后续它们的处理方式与指针类似。

```
leaq    x(%rip), %rax
movq    %rax, -40(%rbp)
```

不能在编译器第一次执行到时获取其存储位置，因为编译器第一次执行到未必是程序第一次执行到

测试

可以通过 lab3 的所有测试文件

存在的问题

- 某些样例可能无法通过
- 产生了许多冗余的 `mov` 等指令
- 代码有些混乱

收获

- 了解了 x86_64 汇编语言，学习使用汇编调试工具（例如 edb）
- 加深了对后端的理解
- 提高了编程能力

备注

Lab5 的代码公开至 <https://github.com/GWDx/Compiler-Cminus>

谢谢观看！